Martin Beckmann, Vanessa N. Michalke, Andreas Vogelsang, Aaron Schlutter

Removal of redundant elements within UML activity diagrams

Suggested Citation
Beckmann, Martin; Michalke, Vanessa N.; Vogelsang, Andreas; Schlutter, Aaron: Removal of redundant elements within UML activity diagrams. - In: 2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS). - New York: IEEE, 2017. ISBN: 978-1-5386-3492-9. - S. 334-343. - DOI: 10.1109/MODELS.2017.7. *(Postprint version is cited, page numbers may differ.)*

Removal of Redundant Elements within UML Activity Diagrams

Martin Beckmann, Vanessa N. Michalke, Andreas Vogelsang, Aaron Schlutter
Technische Universität Berlin, Germany
{martin.beckmann, vanessa.michalke, andreas.vogelsang, aaron.schlutter}@tu-berlin.de

***Abstract*—As the complexity of systems continues to rise, the use of model-driven development approaches becomes more widely applied. Still, many created models are mainly used for documentation. As such, they are not designed to be used in following stages of development, but merely as a means of improved overview and communication. In an effort to use existing UML2 activity diagrams of an industry partner (Daimler AG) as a source for automatic generation of software artifacts, we discovered, that the diagrams often contain multiple instances of the same element. These redundant instances might improve the readability of a diagram. However, they complicate further approaches such as automated model analysis or traceability to other artifacts because mostly redundant instances must be handled as one distinctive element. In this paper, we present an approach to automatically remove redundant *ExecutableNodes* within activity diagrams as they are used by our industry partner. The removal is implemented by merging the redundant instances to a single element and adding additional elements to maintain the original behavior of the activity. We use reachability graphs to argue that our approach preserves the behavior of the activity. Additionally, we applied the approach to a real system described by 36 activity diagrams. As a result 25 redundant instances were removed from 15 affected diagrams.**

I. Introduction

Due to its many advantages [1], model-driven engineering has become a widely applied approach in the development of systems [2]. One of our industry partners (Daimler AG) uses UML2 activity diagrams [3] to specify the functions of systems. Activity diagrams are behavioral diagrams used to create graphical models of stepwise workflows. They are among the types of models, which are regarded beneficial in requirements engineering [4]. A widely applied use of activity diagrams and graphical models in general is to utilize them for communication purposes [5], [6]. Hence, the diagrams are created with a focus on readability and understandability. This is achieved by prioritizing layout aspects of the diagram, since a proper layout is an important factor for understanding the diagrams [7]. As a result, the created diagrams are not catered to be processed by automatic approaches in following stages of development. One of the phenomenons that impede automation are multiple instances of the same element within the activity diagrams. Some of these redundant elements are created intentionally [8] to improve certain aspects of a diagram such as the structure and hence the readability. Other redundant elements arise unintentionally since multiple persons are involved in creating the diagrams. Nonetheless, existing redundant elements complicate possible approaches of automation. For instance, in requirements engineering activities need to be accompanied by textual representations [9]. As a result, a well-structured requirements document should reflect, which executions are possible and necessary in an activity. This relies on the propositional assertions modeled in the activity. To apply simplifications on propositional logic relations (e.g., extract a propositional logic normal form), it is necessary to know, which elements are actually the same. This is guaranteed by a redundancy-free version. Also, for an effective derivation of test cases from activities, path coverage has to be considered. If there are redundant elements, there may be unnecessary paths and hence more test cases are required [10]. In general, checking for path coverage is easier without redundant elements since only unique paths are considered. Other than that, redundancy makes traceability and keeping derived artifacts consistent more difficult.

This paper presents a transformation to remove redundant *ExecutableNode* elements contained within an activity. The removal is achieved by merging all instances of a redundant element into a single instance. In order to preserve the original behavior, new *ControlNodes* are added in the activity. These new *ControlNodes* are connected to the merged element as well as its predecessors and successors. We show that these model transformations preserve the behavior of the original activity by comparing their reachability graphs. We furthermore report on the application of this approach to a set of activity diagrams used to specify a real system from our industry partner. This evaluation shows that redundant elements are common in real activity diagrams and that our approach is able to remove them without blowing up the complexity of the diagrams.

The paper's structure is given in the following. The next section provides details on the situation we encountered at our industry partner. The third section discusses related work on the subject of behavior preserving transformations and of dealing with redundancy in graphical models. In the fourth section, we present the behavior-preserving transformation that removes redundant elements. Section V shows why the transformation preserves the behavior of the activity. Section VI introduces special situations, where less *ControlNodes* are needed for the transformation. In Section VII we analyze activities supplied by our industry partner and present results on applying the transformation on them. Section VIII presents the limits of the approach. The last section concludes this work and gives an outlook on future work.

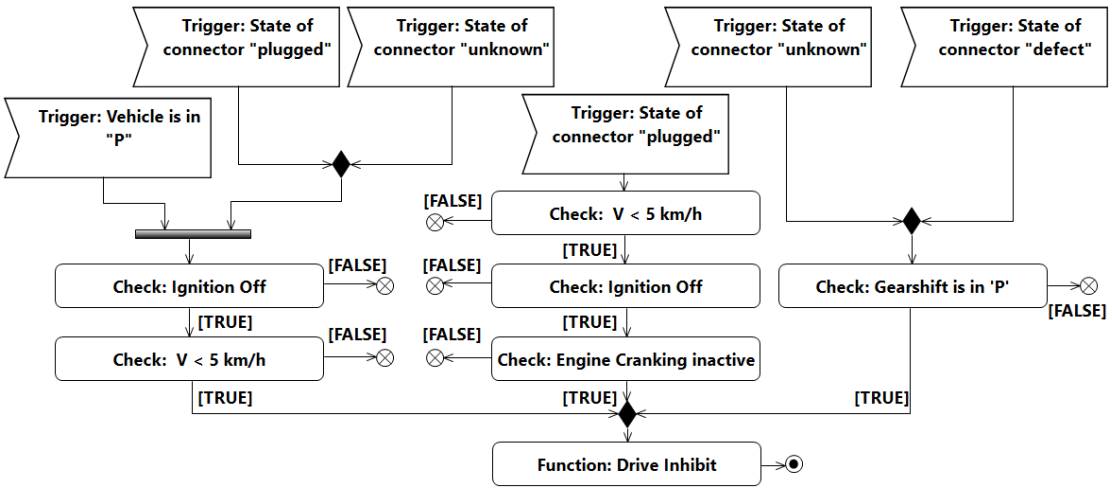


Fig. 1: Exemplar activity diagram containing redundant elements

II. Background

In this section we use an activity diagram provided by our industry partner to show how their diagrams are used and interpreted. In addition, we present our notion of redundancy in activity diagrams.

A. Activity Diagrams Syntax

Our industry partner uses UML activity diagrams as a first step of specifying a new function of a system. An exemplar activity diagram[1] is displayed in Fig. 1. It describes the function's activation and deactivation.

As such, the activity diagram contains a combination of triggers and checks for conditions that must be fulfilled to activate the function. This type of description is similar to Firesmith's proposal for formulating textual natural language requirements [11]. For triggers, the *AcceptEventAction* element is used. Checks are modeled as *Action* elements. If the condition of a check is not fulfilled, the flow ends (*FlowFinal*). The triggers and checks are connected by *ControlNodes* such as *JoinNodes* and *MergeNodes*. *JoinNodes* act as synchronization points and can be interpreted as AND operators in terms of propositional logic. *MergeNodes* represent OR operators. Once the actual functionality of the function is executed, *ActivityFinal* elements designate the end of an activity.

B. Activity Diagram Semantics

We interpret the semantics of activities as Petri net like graphs as suggested by the original UML specification version 2.5 [3, p. 283]. As such, we assume that each *ExecutableNode* executes as soon as a token is placed on that node (by transition or by occurrences of events). Also, we assume that the execution time of the nodes is infinitely fast. This interpretation is related to *requirements-level semantics* for the activity diagram defined by Eshuis and Wieringa [12] and is also used by our industry partner. Furthermore, *ControlNodes* forward tokens instantly if possible. Hence, tokens can be forwarded by multiple *ControlNodes* in one step. Events that execute *AcceptEventActions* of the activity, produce new tokens within the executing activity (i.e., the property *isSingleExecution* is true). Also, we assume that two tokens at an *ExecutableNode* cause two concurrent executions of the *ExecutableNode* within the same step (i.e., the property *isLocallyReentrant* is true).

[1]The displayed activity diagram was slightly modified to incorporate more sophisticated situations.

C. Redundant Elements in Activity Diagrams

In this paper, we are only interested in redundant elements within one diagram and not across different diagrams. The activity diagram, shown in Fig. 1, contains four redundant elements, each having two instances in the diagram. The triggers *State of connector "plugged"* and *State of connector "unknown"* both appear two times. Similar, the two checks $V < 5$ *km/h* and *Ignition Off* also appear twice in the diagram. Elements are considered as redundant elements if they have the same name and the same type (e.g. *AcceptEventAction*). Thus, the considered elements are exact copies of each other apart from their placement within the diagram and their connection to other elements, which makes these elements **Type A Clones** according to Störrle's classification [13].

While this duplication of the same element increases the number of elements in the diagram, it may also increase the comprehensibility of the diagram. For instance, the duplicated elements in Fig. 1 allow the visual separation of three distinct possibilities, that lead to the function's activation.

In a previous work, we defined and analyzed different types of quality issues that arise when activity diagrams are used in requirements documents for the specification of functions [14]. One of the quality issues, we identified, are redundant elements. In that study, we found redundant elements in more than 40% of the examined diagrams, which shows that this is a common phenomenon. On the other hand, developers rated the appearance of redundant elements as one of the least severe quality issues. However, this work solely focuses on enabling the use of the diagrams for automation rather than creating an alternative view for existing diagrams.

III. Related Work

Since our approach transforms models in a behavior preserving way, it is related to refactoring [15]. In contrast to classical refactoring, our aim is not to improve the design but to facilitate the processing by automated techniques.

Refactoring of UML models has been covered by a number of publications [16]. Focal point of their research is the UML class diagram as it is the most used UML diagram [17]. Among others, examples for refactorings of class diagrams are presented in [18], [19], [20], [21]. Another type of diagram that has received attention in relation to refactoring are statecharts [18], [21]. For activity diagrams, two refactoring operations are described in [21], namely *Make Actions Concurrent* and *Sequentialize Concurrent Actions*. As these names of the operations indicate, they are not suitable to deal with redundant elements. A more extensive review on refactoring UML models can be found in [22]. An approach of detecting semantically equivalent modeling concepts for structurally different models is described in [23].

Besides the transformation of models, redundancy in UML models has also been a topic in research, although the focus is mainly placed on the detection of redundancy (see [24] for a list of approaches). For Petri nets, as a basis for activity diagrams, the elimination of redundant control places while keeping a Petri net live is described by Uzam et al. [25]. In addition to presenting an approach to detect clones in models, Störrle gives an example of a transformation that removes recurring fragments of activities by factoring them into independent activities [13]. The main rationale behind these refactoring proposals is to increase the maintainability of models and reduce the risk of inconsistent changes. Our work, in contrast, deals with the removal of redundant elements within a single activity for the purpose of facilitating their processing by automatic approaches. This also includes elements that UML defines as "integral parts of a diagram" (such as *DataFlowNodes*), which Störrle calls *loophole clones* and for which his refactoring approach does not work. It is also mentioned, that there are tools distinguishing between internal representation (the activity itself) and an external visual representation (the activity diagram). Most contemporary tools enforce a one-to-one correspondence between these two representation types. This simplifies, the handling of copy/paste operations [13]. As a consequence, using an element of the internal representation multiple times in the external representation is not possible. A tool, that does not enforce one-to-one correspondence would allow for proper readability while still allowing for automatic approaches. Nonetheless, even if such a tool is used, the modeler must still be aware of this capability and is required to consider this fact during model creation. Hence, a tool-independent approach is needed, that takes into account the modeling-process and its challenges.

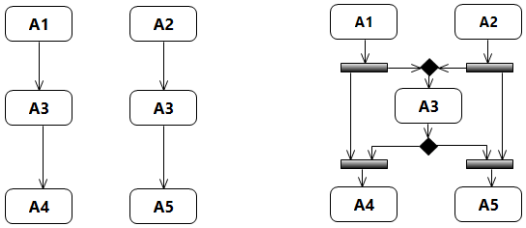
Activities can be the source for a number of possible applications. Among others they are used to automatically generate textual specification documents [26], source code [27] and test cases [28].

IV. Elimination of Redundant Elements

A. Transformation

To remove redundant *ExecutableNode* elements from an activity, we propose a transformation that consists of three steps. All steps have to be performed for every redundant element in an activity. The first step adds a new element, that represents all instances of the redundant element. The second step adds *ControlNodes* to the activity. The predecessors and successors of the instances and the added element are connected with the *ControlNodes* by *ControlFlows*. Lastly, all instances of the considered redundant element are removed from the activity. As a result the element added in the first step remains as a single instance of the redundant element.

The necessary *ControlNodes* are *ForkNodes*, *JoinNodes*, *MergeNodes* and *DecisionNodes*. A single *MergeNode* is added as a predecessor to the remaining element. A single *DecisionNode* is added as a successor to the remaining element. For each instance of the redundant element one *ForkNode* and one *JoinNode* is added. The *ForkNodes* are predecessors to the added *MergeNode*. The predecessor of each *ForkNode* are the predecessors of the original instances. The *JoinNodes* are successors of the added *DecisionNode*. The successors of each *JoinNode* are the successors of the original instances. Each of the added *ForkNodes* has an outgoing edge to an added *JoinNode*. Thereby, the *ForkNode*, which is added as the predecessor to one instance is connected to the *JoinNode*, which is added as the successor to the same instance. Since there are no guards on the outgoing edges of the *DecisionNode*, an incoming token is forwarded to a *JoinNode* with a token present [3, p. 373, p. 387].



(a) Original activity (b) Transformed activity

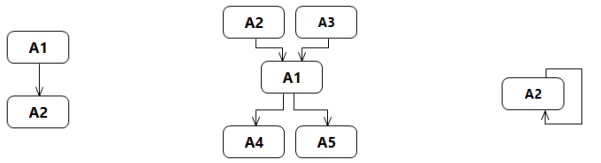
Fig. 2: Example of the transformation

Fig. 2 shows an activity diagram fragment before (Fig. 2a) and after (Fig. 2b) the transformation. The depicted activity has the redundant element A3 with the two redundant instances $A3_1$ and $A3_2$. The instances are denoted with indices for distinction. The *Actions* $A1$, $A2$, $A4$ and $A5$ can be any *ExecutableNode* (e.g. *Actions*, *AcceptEventActions*) or multiple *ExecutableNodes* or *ControlNodes*. In case *A1* executes, an execution of $A3$ follows. Because of the execution of $A1$, there is a token present at the *JoinNode* before *A4*. The token produced by $A3$ is forwarded to this *JoinNode*. Thus $A4$ executes. In case both $A1$ and $A2$ are executed, $A3$ is executed two times and hence produces two tokens, which lead to the execution of $A4$ and $A5$. In both cases, it is the same behavior as before.

B. Normal Form

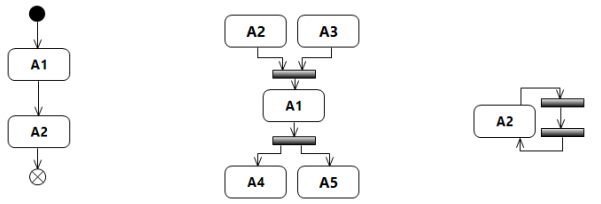
The transformation is applicable if all instances of the redundant elements have a single predecessor and a single successor. For this purpose, we denote an activity with redundant elements, where each of its instances has exactly one predecessor and one successor, to be in a normal form. If this is not the case, further *ControlNodes* are added to make sure this requisite is fulfilled.

An activity is not in the normal form if one of its instances is missing a predecessor or a successor. If the predecessor is missing, an *InitialNode* is added as a predecessor [3, p. 376]. If the successor is missing, a *FlowFinal* is added as a successor since the flow of tokens ends after the *ExecutableNode*. In Fig. 3a a situation is displayed, where one element does not have a predecessor and one element does not have a successor. Fig. 4a shows the corresponding situation with an added *InitialNode* and *FlowFinal*.



(a) No predecessor / successor (b) Multiple predecessors / successors (c) Cycle

Fig. 3: Situations without normal form



(a) No predecessor / successor (b) Multiple predecessors / successors (c) Cycle

Fig. 4: Situations of Fig. 3 in normal form

In addition to missing predecessors or successors, there might be instances with more than one predecessor or successor. If there are multiple predecessors, a *JoinNode* is added to make the implicit *Join* to an explicit *Join* [3, p. 401]. After the transformation, this results in a *JoinNode* with an outgoing edge to the respectively added *ForkNode* before the remaining element. If there are multiple successors, a *ForkNode* is added to make the implicit *Fork* to an explicit *Fork* [3, p. 401]. After the transformation, this results in a *ForkNode* with an incoming edge from the respectively added *JoinNode* after the remaining element. An example of the described situation is displayed in Fig. 3b. The corresponding, for Action A1 resolved situation is shown in Fig. 4b.

An instance of a redundant element can also be a part of a cycle. In case the instance is its own predecessor and successor, it is necessary to add *ControlNodes* as new predecessors and successors. As a predecessor a *ForkNode* is added and as a successor a *JoinNode* is added. In Fig. 3c such a situation is displayed. Fig. 4c shows the corresponding situation with an added *JoinNode* and *ForkNode*. It is also possible to add a *DecisionNode* and a *MergeNode* or any other combination of *ControlNodes*, as these *ControlNodes* only have one incoming and one outgoing edge. However a *ForkNode* and a *JoinNode* can be merged, with the respective *JoinNode* and *ForkNode*, that are added by the transformation.

By using *ControlNodes* with one incoming and outgoing edge, ultimately every combination of predecessors and successors of the redundant elements can be converted to the normal form.

C. Number of additional elements

The number of necessary additional *ControlNodes* and *ControlFlow* edges depends on the number of redundant elements and on how many instances are part of each redundant element. There is a new *ForkNode* and a new *JoinNode* for each instance. Also, there is one additional *MergeNode* and one additional *DecisionNode* for each redundant element. Besides, there might be *ControlNodes* necessary to ensure, that each instance has a single predecessor and successor. Thus, the number of new *ControlNodes* in an activity results in:

$$\#NewControlNodes = \sum_{i=1}^{m}(2 * n_i + 2) + c \qquad (1)$$

The variable m denotes the number of redundant elements, n_i denotes the number of instances in each redundant element and c denotes the number of *ControlNodes* needed for the normal form. Since, there is a maximum of two *ControlNodes* needed for every instance to create the normal form, there is a linear relation between the number of redundant elements and the additionally needed *ControlNodes*.

Using the same notation and, additionally, the variable d as the number of needed edges, the number of new *ControlFlow* edges in an activity results in:

$$\#NewControlFlowEdges = \sum_{i=1}^{m}(3 * n_i + 2) + d \qquad (2)$$

For every instance of a redundant element, there need to be three additional edges (two outgoing edges of the added *ForkNode*, one incoming edge of the added *JoinNode*). For every redundant element two edges are needed as the outgoing edge of the *MergeNode* and the incoming edge of the *DecisionNode*. Additionally, there is a maximum of two edges needed for each instance of an redundant element, to create the normal form. Hence, there is also a linear relation between the number of redundant elements and the additionally needed *ControlFlow* edges. The linear relations for the number of nodes and edges are important, since an automated processing might be impaired otherwise.

D. Resulting Structure

The usage of a single predecessor and a single successor results in a single-entry single-exit structure for the transformed part. As a consequence everything before and after the remaining element stays unchanged. The principle of compositionality applies. This means, that the behavior of the activity remains the same, if the behavior of the changed part remains the same. Hence, to show the preservation of the behavior, it is sufficient to show, that the behavior of the transformed part stays the same.

V. Preservation of Behavior

To show that the transformation preserves the behavior expressed in an activity diagram, we compare the flow of tokens in the underlying semantic model (see Section II). Reachability graphs (RG) [29] represent this flow of tokens in a network depending on the executed actions. Hence, we use reachability graphs as a means to show, that the behavior of the activity, before and after applying the transformation, is still the same. In this chapter, we briefly introduce reachability graphs. Additionally, we argue why the comparison of the RGs of the activities is suitable to show the preservation of the behavior. Subsequently, we propose a way to derive RG from activities. In the subsection after that, we show how to compare two activities by using RGs.

A. Reachability Graphs

We construct the reachability graph RG for an activity A as:

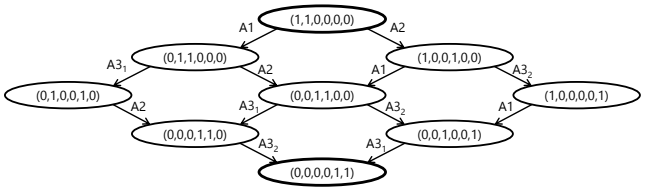
$$RG(A) = (M(V), E) \tag{3}$$

V is the set of *ActivityNodes* contained in the activity A. $M(V)$ is the set of distributions of tokens to the *ActivityNodes*. Thus, every node $m \in M(V)$ in the RG represents a distribution of tokens within the activity A. Every element of $M(V)$ is a $|V|$-tuple, where each entry represents the number of tokens at every *ActivityNode* after a sequence of executions. E is the set of directed edges of the RG. The edges represent the execution of an *ExecutableNode*, which leads to a new distribution of tokens.

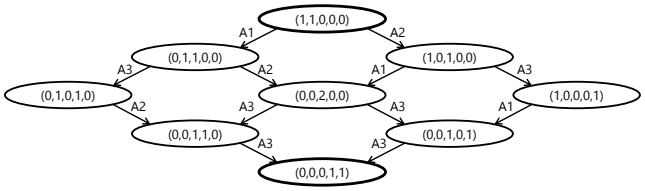
The *initial distribution* $m_0 \in M(V)$ represents the distribution of tokens at the beginning of execution. Its node in the RG has no incoming edges. The *initial distribution* depends on the events that might occur at the beginning and on existing *InitialNodes*. There may be multiple different *initial distributions*, which each result in different RGs. The *final distribution* $m_n \in M(V)$ represents the distribution of tokens in the activity, where no more nodes are left to be executed. This is also the case as soon as one token reaches an *ActivityFinal*. Its node in the RG has no outgoing edges. Each RG has only one *final distribution*.

A *sequence of executions* is a sequence of edges $(e_1, ..., e_n)$ in the reachability graph RG, which starts at the *initial distribution* and ends at the *final distribution*. Hence, a *sequence of executions* represents a possible order of executed *Actions* in the activity that lead from the *initial distribution* to the *final distribution*.

Fig. 5a shows the RG of the activity presented in Fig. 2a. The RG of the transformed activity from Fig. 2b is displayed in Fig. 5b. In both cases, it is assumed that the *initial distribution* results from the events executing $A1$ and $A2$ simultaneously. Also, the distributions in both figures only incorporate *ExecutableNodes*.



(a) RG $(A1, A2, A3_1, A3_2, A4, A5)$ for the original activity



(b) RG $(A1, A2, A3, A4, A5)$ for the transformed activity

Fig. 5: RGs for the activities in Fig. 2

While the distribution of tokens in Fig. 5a is represented by a 6-tuple, the distribution in Fig. 5b is represented by a 5-tuple. This results from the different number of *ExecutableNodes* in the two activities, since the two redundant *Actions* $A3_1$ and $A3_2$ were replaced by $A3$.

An RG contains all possible *sequences of executions* of an activity for a given *initial distribution*. As a consequence, we conclude that if the same sequences lead to the same distribution of tokens in an activity, then the behaviors of the activities are same. As a result, the comparison of two RGs of the respective activities shows the preservation of the behavior for a given *initial distribution*.

B. Generating Reachability Graphs from Activity Diagrams

RGs are generated for a chosen *initial distribution*. Hence, the first step is to decide how many tokens are initially placed on each *ActivityNode*. The resulting distribution of tokens is the first node (the *initial distribution*) of the RG.

From the *initial distribution*, the RG is constructed step by step. The underlying algorithm is basically the same as for a Petri net. For every entry in the current distribution, which has at least one token, the token is transferred from the corresponding *ActivityNode* to its successor in the activity. This results in a new distribution in the RG, which is connected by an incoming edge to the previous distribution in the RG. Although the *ControlNodes* do not hold tokens, they are included as entries in the distributions. This is necessary, since they may change the number of tokens. As a result the number

of tokens of a new distribution depends on the type of the executed *ActivityNode*. If the current *ActivityNode* is an *ExecutableNode*, every token is forwarded to the successor after the *ExecutableNode* is executed (assuming that there are no implicit *ControlNodes*). The different types of *ControlNodes* on the other hand all show a different behavior towards the number of tokens in the activity. Hence, every *ControlNode* needs to be considered differently. In the following, $t(v)$ denotes the number of tokens at a certain *ActivityNode* $v \in V$.

MergeNode. *MergeNodes* forward tokens from multiple incoming *ActivityEdges*. As such, they act as OR-connections between the predecessors. As a consequence, for a given distribution $(..., t(v), ..., t(v'), ...)$, where $v \in V$ is the *MergeNode* and $v' \in V$ is a successor, the distribution $(..., t(v) - 1, ..., t(v') + 1, ...)$ is added as a node in the RG.

ForkNode. *ForkNodes* pass a single token to each outgoing *ActivityEdge* for each token on the incoming *ActivityEdge*. For a given distribution $(..., t(v), ..., t(v_1), ..., t(v_n), ...)$, where $v \in V$ is the *ForkNode* and $v_1, ..., v_n \in V$ are the following nodes on the outgoing *ActivityEdge* of the *ForkNode*, results the distribution $(..., t(v) - 1, ..., t(v_1) + 1, ..., t(v_n) + 1, ...)$.

JoinNode. *JoinNodes* act as AND-connections as they only forward a single token on their outgoing *ActivityEdge*, if there is one token present at each incoming *ActivityEdge*. For a given distribution $(..., t(v), ..., t(v'), ...)$, where $v \in V$ is the *JoinNode* and $v' \in V$ is the successor, if there are $n \in \mathbb{N}$ incoming *ActivityEdges*, where each incoming *ActivityEdge* has a single token present, the distribution $(..., t(v) - n, ..., t(v') + 1, ...)$ is created.

DecisionNode. *DecisionNodes* forward a single token to an applicable outgoing *ActivityEdge* if there is a token on the incoming *ActivityEdge*. If there is more than one *ActivityEdge* applicable, the token only traverses one of the *ActivityEdges* [3, p. 388]. Since there are different RGs depending on the decisions, there has to be a new RG for every possible decision and not just a new distribution. For a given distribution $(..., t(v), ..., t(v_1), ..., t(v_i), ..., t(v_n), ...)$, where $v \in V$ is the *DecisionNode*, $v_1, ..., v_n \in V$ are successors of the *DecisionNode* and v_i is the node, that accepts the token offered by the *DecisionNode*, follows the new distribution $(..., t(v) - 1, ..., t(v_1), ..., t(v_i) + 1, ..., t(v_n), ...)$.

C. Comparison of Activity Diagrams

For a complete comparison of the behavior of two activities, one would need to compare all possible RGs of both activities. Different RGs arise from different *initial distributions* and non-deterministic *DecisionNodes* [3, p. 387]. Due to the unlimited number of tokens that may be placed in the *initial distribution*, there is an infinite number of possible RGs. Therefore, in general, it is not possible to compare all RGs. We argue that it is sufficient to compare just the RGs with all possible combinations of *initial distributions*, where each suitable *ActivityNode* has either no token at all or just one token. The reason is that all combinations of zero or one token represent all possible flows in the activity. This rationale ignores the fact that there might be structures in an activity that require a certain number of tokens. Suitable *ActivityNodes* are those that might have a token at the start of the execution of an activity (*InitialNodes* and *AcceptEvenActions*). For n suitable *ActivityNodes*, this results in $2^n - 1$ different RGs for different *initial distributions*. The distribution of no tokens at all does not yield any information regarding the behavior.

These RGs are needed for the comparison of the behavior of two activities. The actual comparison of two RGs is done with two RGs representing the same situation, i.e., the same *initial distribution* and the same decisions made. One criterion for two RGs to be equivalent is that their *sequences of execution* are equivalent. This means, that every *sequence of execution* in the RG of the original activity has an equivalent *sequence of execution* in the RG of the transformed activity. As the *ControlNodes* do not hold tokens and do not execute operations other than manipulating the flow of tokens, they must not be considered during the comparison of two RGs. Note that there are redundant sequences of executed *Actions* in the RGs because of the redundant elements.

Besides the *sequences of execution*, the equality of the individual distributions is the second criterion that must be fulfilled. Otherwise, the behavior is not the same if the same sequences lead to different distributions. The transformed activity contains less entries in the distributions of the RG than the original activity (see subsection V-A) because of the removed redundant elements. Thus, the sum of tokens of the redundant elements equals the number of tokens of the remaining elements. For the distribution $(y_i, ..., y_{m-(n-i)})$ of the transformed activity an equivalent distribution $(x_1, ..., x_i, ..., x_n, ..., x_m)$ of the original activity, given redundant instances x_i to x_n, can be identified by:

$$y_k = \begin{cases} x_k & if\ k = [1, i-1] \\ \sum_{l=i}^{n} x_l & if\ k = i \\ x_{k+(n-i)} & if\ k = [i+1, m-(n-i)] \end{cases} \quad (4)$$

Because of the single-entry single-exit structure of the transformation (see subsection IV-A), it is only necessary to compare the changed part of the activity before and after the transformation. Since the transformation introduces pairs of *ForkNodes* and *JoinNodes* for the respective predecessors and successors of the original instances, the forwarding of tokens by the introduced *DecisionNodes* are deterministic and hence do not require additional RGs. The deterministic behavior results from the fact that it is always clear to which *JoinNode* a token is forwarded in each step. In the example Fig. 2a, there are two suitable *ActivityNodes* for the *initial distribution*. Thus, three pairs of RGs need to be compared.

The *sequences of executions* of the RG in Fig. 5a are the following:

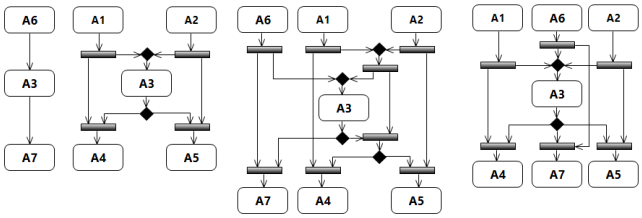
1) $A1, A3_1, A2, A3_2$
2) $A1, A2, A3_1, A3_2$
3) $A1, A2, A3_2, A3_1$
4) $A2, A1, A3_1, A3_2$
5) $A2, A1, A3_2, A3_1$
6) $A2, A3_2, A1, A3_1$

The *sequences of executions* of the RG in Fig. 5b are:

1) $A1, A3, A2, A3$
2) $A1, A2, A3, A3$
3) $A1, A2, A3, A3$
4) $A2, A1, A3, A3$
5) $A2, A1, A3, A3$
6) $A2, A3, A1, A3$

Since *Action* $A3_1$ and $A3_2$ in the original activity in Fig. 2a are redundant instances of the same redundant element, the *sequences of executions* 2) and 3) as well as 4) and 5) are the same. By removing the redundant *sequences of executions* and comparing the remaining ones, it follows that both RGs contain the same *sequences of executions*.

Considering the criteria defined in Equation 4 for the equality of the distributions, the distributions are equal as well. As a result, the RGs in Fig. 5 are isomorphic. The same holds for the RGs generated by using the other two *initial distributions*. Hence, for the assumed semantics, the transformation preserves the behavior if it is applied to an activity containing one redundant element consisting of two instances. The transformation can also be applied to activities with more than one redundant element and with more than two instances. This is based on the fact that the transformation can be applied in any order. From this follows that the removal of multiple instances can be conducted by applying the transformation to only two instances each time until there is one instance left. Additional *ControlNodes* resulting from the consecutive application of the transformation can be merged.



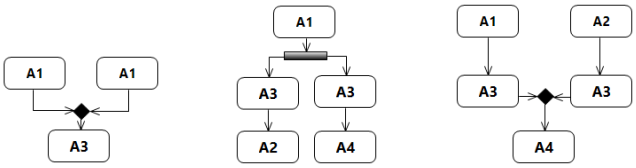
(a) Initial Situation (b) Applied Transformation (c) Merged *ControlNodes*

Fig. 6: Transformation of a redundant element with three instances

The consecutive application of the transformation is shown in Fig. 6. In the initial situation in Fig. 6a, there is a redundant element with two instances. The structure of the instance on the right hand side, results from a previous application of the transformation on two instances. If the transformation is applied in this situation the *ExecutableNode A6* and the added *MergeNode* are used as the predecessors and the *ExecutableNode A7* and the added *DecisionNode* are used as successors. This results in the activity displayed in Fig. 6b. This structure can be simplified to the structure displayed in Fig. 6c without changing the flow of tokens. Since the added *MergeNode* and *DecisionNode* are now the new predecessor and successor, this procedure can be repeated if there are further instances.

VI. Special Situations

Besides the presented transformation, there are special situations where the removal of the redundant elements is possible using less additional *ControlNodes*. Three examples are shown in Fig. 7.

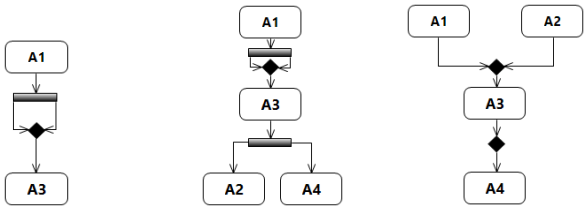


(a) Situation 1 (b) Situation 2 (c) Situation 3

Fig. 7: Three special situations with redundant elements

The activity in Fig. 7a contains two redundant elements. Both elements do not have a predecessor. The activity in Fig. 7b contains two redundant elements, which have a common predecessor and distinct successors. The activity in Fig. 7c contains two redundant elements, which have distinct predecessors and a common successor.

Their respective activities without redundant elements are depicted in Fig. 8.



(a) Situation 1 (b) Situation 2 (c) Situation 3

Fig. 8: Three special situations without redundant elements

In all of these situations there is only one of the two redundant instances left. Compared with the generic transformation introduced in subsection IV-A, less additional *ControlNodes* and *ControlFlow* edges are needed. Hence, the resulting number of *ControlNodes* and of *ControlFlow* edges in Equation 1 and in Equation 2 respectively are upper limits.

The preservation of behavior for these activities results from the fact that the missing predecessors and successors lead to structures that do not change the flow of the tokens. For instance, there are *ControlNodes* without incoming edges or *ControlNodes* with a single incoming and a single outgoing edge. We additionally verified the preservation of behavior by constructing the RGs for these activities. The resulting RGs are equivalent for the necessary distributions.

A. Introductory Example Revisited

If the presented approach is applied to the introductory example in Fig. 1, this results in the activity diagram displayed in Fig. 9. As the activity is meant as a source for further automated approaches, the activity diagram is displayed to illustrate the applied transformations. To increase the readability, we left out the *FlowFinal* elements and not all of the

implicit connections are depicted explicitly. As there are four redundant elements with two instances each, four elements are removed. For the redundant triggers, the transformation for situation 1, presented in Section VI is applied. For the redundant checks the generic transformation is applied. In contrast to the original activity, the number of *ControlNodes* increases and intersecting edges appear. This results in decreased readability, which makes the activity harder to understand. Since the aim of the transformation is to improve applicability of automated approaches, the decreased readability is not an issue within the scope of this work. Hence, we propose to use the redundancy-free activity as a parallel artifact.

VII. Evaluation

To evaluate the applicability of our approach in activity diagrams created in practice, we applied the approach to the activity diagrams of a system of an industry partner. The system's functions were specified by a total of 36 activity diagrams (containing between 9 to 28 *ExecutableNodes*). Each activity describes a function of a system, which is responsible for charging the high-voltage batteries of Plug-in Hybrid Electric Vehicles and Battery Electric Vehicles. As such the system contains functions that are relevant for safety as well as for usability. The aim of the evaluation is to answer the following questions:

- **RQ1:** How many redundant elements appear in activities of a real system?
- **RQ2:** Is our transformation approach applicable to every situation in the activities of a real system?
- **RQ3:** How many additional elements are introduced when applying the transformations?
- **RQ4:** How many of the special situations occur in the activities of a real system?

A. Implementation

The original data of our industry partner was supplied as an Enterprise Architect project file. To apply the transformation, two steps were required to prepare the data. The first step is to convert the project file (*.eap*) to a *.uml* file. The conversion was done automatically by a self-written converter. Our implementation of the transformation approach is realized using *.uml* files because, in contrast to, e.g., the *.eap* format, the *.uml* data format is aligned with the UML specification. Hence, it is only necessary to adjust the converter if a different data source is used in future. In the second step, the resulting *.uml* file is edited manually. This is necessary if our converter encountered situations that it could not handle. Such situations may result from deviations between the Enterprise Architecture data model and the *.uml* data model. Another reason for the manual adjustments was to correct the use of a number of *ControlNodes* originating from developer mistakes and misunderstandings (e.g. *MergeNodes* and *DecisionNodes* as well as *JoinNodes* and *ForkNodes* were sometimes mixed up because they look the same). The transformation itself was implemented to work on the resulting *.uml* files.

TABLE I: Extent of redundant elements in the analyzed system.

Finding	**#Diagrams**	**Ratio**
Total activity diagrams	36	100%
Containing redundant elements	15	42%
Containing 1 redundant element	8	19%
Containing 2 redundant elements	5	17%
Containing 3 redundant elements	2	6%
Finding	**#Red. Elements**	**Ratio**
Total redundant elements	24	100%
Containing 2 instances	23	96%
Containing 3 instances	1	4%

TABLE II: Results of applying the transformation

Finding	**Number**
Removed *ExecutableNodes*	25
Added *ControlNodes* max.	146
Added *ControlNodes* min.	111
Generic Transformation	18
Situation 1	4
Situation 2	0
Situation 3	3

B. Study Results

To answer the first question, we analyzed the activities towards the number of occurrences of redundant elements. The detailed results are displayed in Table I.

Out of the 36 diagrams, we found 15 diagrams containing redundant elements. The 15 diagrams contain 24 elements that appear multiple times in each activity diagram. Of these 15 diagrams, there are 2 diagrams each containing 3 redundant elements with two instances. Another 5 diagrams contain 2 redundant elements. Out of these 5 diagrams 4 diagrams have redundant elements with two instances each. The fifth diagram contains a redundant element with three instances as well as one with two instances. The remaining 8 diagrams contain only one redundant element with two instances each.

The transformation was applicable to all provided activities. There is no constellation, where the transformation would not preserve the behavior. An analysis of the results of the applied transformations is shown in Table II.

The removal of the 23 redundant elements with two instances and the one redundant element with three instances results in an overall of 25 removed *ExecutableNodes*. When only applying the generic transformation, a total of 146 *ControlNodes* were added (*Added ControlNodes max.*) to the activities. However, when also using the smaller transformations for the special situations explained in section VI, the generic transformation only had to be applied 18 times. Situation 1 was applicable four times and situation 3 was applicable three times. The special situation 2 did not occur. By utilizing the

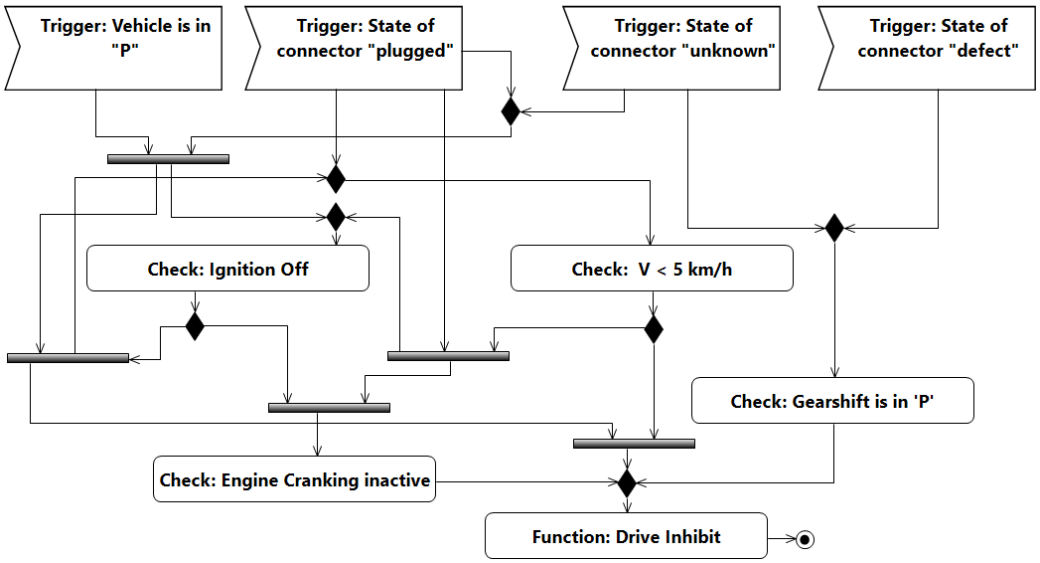


Fig. 9: Introductory example without redundant elements

transformations of the special situations, only 111 additional *ControlNodes* had to be added.

It has to be noted, that we only examined one system of one single industry partner. As a result, the generalizability of our findings is limited in regard to whether the approach is always applicable and whether the resulting numbers are representative.

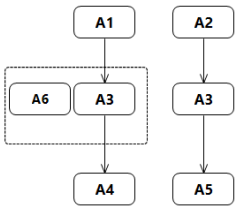
VIII. Limitations of the Approach

The presented approach is restricted to our interpretation of the semantics of an activity. If different semantics are used, the transformation might no longer preserve all aspects of the behavior. Assuming a semantic where every *ExecutableNode* has its own individual execution time and events can occur at any time, this might lead to tokens overtaking one another. Hence, the order in which the actions are executed is no longer the same. Still, the same *Actions* are executed the same number of times. A possible sequence of executions of the activity in Fig. 2a is shown in 1). A possible deviating sequence of executions of the redundancy-free activity in Fig. 2b is shown in 2).

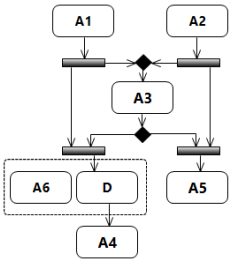
1) $A1, A2, A3_1, A3_2, A4, A5$ 2) $A1, A2, A3, A3, A5, A4$

For instance, while $A1$ is executing, $A2$ starts executing. As soon as $A1$ finishes, $A3$ starts execution. If $A2$ finishes execution while $A3$ is still running, then both *JoinNodes* have one token present. As a result of the non-deterministic behavior of the *DecisionNode*, both $A4$ and $A5$ are able to accept the token produced by the first execution of $A3$. Hence, instead of executing $A4$ as in the original activity, $A5$ might be executed. In this case, the second execution of $A3$ results in the execution of $A4$.

Besides, the presented approach is limited to a subset of available elements in activities. The behavior might not be



(a) Original activity



(b) Transformed activity

Fig. 10: Redundant element in an *InterruptibleRegion*

preserved in case other elements of activities are used (e.g., guards). In Fig. 10a, an activity is displayed where one instance of the redundant element is part of an *InterruptibleRegion*. As soon as the execution of *A3* ends, the execution of $A6$ ($A6$ is a substitution for multiple elements in the *InterruptibleRegion*) is also ended. If the transformation is applied and the remaining element stays part of the *InterruptibleRegion*, $A6$ is always terminated no matter, which predecessor was executed before $A3$. If the remaining element is no longer part of the *InterruptibleRegion*, $A6$ is no longer terminated by the execution of $A3$ as a successor of $A1$. A possible way to resolve this, is shown in Fig. 10b. By introducing a dummy *ExecutableNode* D as a successor to $A3$ before $A4$ and putting the dummy node in the *InterruptibleRegion*, it is still possible to maintain the original behavior. Since this transformation involves an additional *ExecutableNode*, we do not consider this a part of our proposed transformation. Aside from *InterruptibleRegions*, there might be other constellations of elements in activities, where the approach does not preserve the behavior either.

IX. Conclusion and Future Work

By investigating a set of UML2 activity diagrams from an industry partner, we showed, that there are activities in practical use, that contain a number of redundant elements. To improve the use of these activities for automated approaches, we proposed a transformation that removes redundant elements while preserving their behavior. The transformation and its property of behavior preservation are based on the assumption of Petri net based semantics. The transformation merges the redundant elements to a single element, adds *ControlNodes*, and connects them to existing elements to assure the preservation of behavior. The number of added *ControlNodes* is in a linear relation to the number of redundant elements. There are special cases that need less *ControlNodes* for the preservation. In order to show the preservation of behavior after transforming the activities, we presented how to derive reachability graphs from an activity and how to compare reachability graphs of different activities. The comparison showed that the transformation preserves the behavior of an activity containing multiple redundant elements with multiple instances. Since the transformation creates a single-entry single-exit structure, we argue that the preservation is valid in general.

Although we argue for the preservation of the behavior, a formal proof for correctness is still needed. There are a number of other formal semantics proposed for UML2 activities. Whether or not all aspects of the preservation hold for these semantics is worth investigating as well as considering all possible constellations of elements in activities.

References

[1] L. Apfelbaum and J. Doyle, "Model Based Testing," in *Software Quality Week Conference*, 1997.

[2] J. Hutchinson, M. Rouncefield, and J. Whittle, "Model-Driven Engineering Practices in Industry," in *33rd International Conference on Software Engineering (ICSE)*, 2011.

[3] Object Management Group (OMG), "OMG Unified Modeling Language (OMG UML), Version 2.5," OMG Document Number formal/2015-03-01 (http://www.omg.org/spec/UML/2.5/), 2015.

[4] E. Sikora, B. Tenbergen, and K. Pohl, "Industry needs and research directions in requirements engineering for embedded systems," *Requirements Engineering*, vol. 17, no. 1, 2012.

[5] M. Brambilla, J. Cabot, and M. Wimmer, "Model-Driven Software Engineering in Practice," *Synthesis Lectures on Software Engineering*, 2012.

[6] D. Drusinsky, "From UML activity diagrams to specification requirements," in *IEEE International Conference on System of Systems Engineering (SoSE)*, 2008.

[7] H. Störrle, "On the Impact of Layout Quality to Understanding UML Diagrams," in *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*. IEEE, 2011.

[8] W. Liu, S. Easterbrook, and J. Mylopoulos, "Rule-Based Detection of Inconsistency in UML Models," in *Workshop on Consistency Problems in UML-Based Software Development*, vol. 5, 2002.

[11] D. Firesmith, "Generating Complete, Unambiguous, and Verifiable Requirements from Stories, Scenarios, and Use Cases." *Journal of Object Technology*, vol. 3, no. 10, 2004.

[9] M. Beckmann and A. Vogelsang, "What is a Good Textual Representation of Activity Diagrams in Requirements Documents?" in *7th International Model-Driven Requirements Engineering Workshop (MoDRE)*, 2017.

[10] H. Kim, S. Kang, J. Baik, and I. Ko, "Test Cases Generation from UML Activity Diagrams," in *8th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD)*, vol. 3, 2007.

[12] R. Eshuis and R. Wieringa, "Tool Support for Verifying UML Activity Diagrams," *IEEE Transactions on Software Engineering*, vol. 30, no. 7, 2004.

[13] H. Störrle, "Towards clone detection in UML domain models," *Software & Systems Modeling*, vol. 12, no. 2, 2013.

[14] M. Beckmann, A. Vogelsang, and C. Reuter, "A Case Study on a Specification Approach using Activity Diagrams in Requirements Documents," in *25th IEEE International Requirements Engineering Conference*, 2017.

[15] M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[16] T. Mens, "On the Use of Graph Transformations for Model Refactoring," in *Generative and transformational techniques in software engineering*. Springer, 2006.

[17] T. Mens, G. Taentzer, and D. Müller, "Model-Driven Software Refactoring," *Model-Driven Software Development: Integrating Quality Assurance*, 2008.

[18] G. Sunyé, D. Pollet, Y. Le Traon, and J.-M. Jézéquel, "Refactoring UML Models," in *International Conference on the Unified Modeling Language*. Springer, 2001.

[19] A. Correa and C. Werner, "Applying Refactoring Techniques to UML/OCL Models," in *International Conference on the Unified Modeling Language*. Springer, 2004.

[20] M. Van Kempen, M. Chaudron, D. Kourie, and A. Boake, "Towards Proving Preservation of Behaviour of Refactoring of UML Models," in *Research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*. South African Institute for Computer Scientists and Information Technologists, 2005.

[21] M. Boger, T. Sturm, and P. Fragemann, "Refactoring Browser for UML," in *International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World*. Springer, 2002.

[22] M. Misbhauddin and M. Alshayeb, "UML model refactoring: a systematic literature review," *Empirical Software Engineering*, vol. 20, no. 1, 2015.

[23] K. Altmanninger, "Models in Conflict–Towards a Semantically Enhanced Version Control System for Models," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2007.

[24] B. Kaur and E. H. Kaur, "Clone Detection in UML Sequence Diagrams Using Token Based Approach," *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 5, no. 5, 2015.

[25] M. Uzam, Z. Li, and M. Zhou, "Identification and elimination of redundant control places in Petri net based liveness enforcing supervisors of FMS," *The International Journal of Advanced Manufacturing Technology*, vol. 35, no. 1, 2007.

[26] J. Nicolás and A. Toval, "On the Generation of Requirements Specifications from Software Engineering Models: A Systematic Literature Review," *Information and Software Technology*, vol. 51, no. 9, 2009.

[27] M. Usman and A. Nadeem, "Automatic Generation of Java Code from UML Diagrams using UJECTOR," *International Journal of Software Engineering and Its Applications*, vol. 3, no. 2, 2009.

[28] D. Kundu and D. Samanta, "A Novel Approach to Generate Test Cases from UML Activity Diagrams." *Journal of Object Technology*, vol. 8, no. 3, 2009.

[29] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, 1989.